An Implementation of a File System in User Space using the FUSE framework and Telegram

Coello Andrade Mateo David[1][0000−0002−2201−0256] and Mendoza Nuñez Patricio Joshue[2][0000−0001−8919−2579]

Yachay Tech University, Urcuquí, Ecuador
{mateo.coello, patricio.mendoza}@yachaytech.edu.ec

Abstract. This report develops the notion of file system in user space, first introducing the concept of file system architecture and continuing with an explanation of the virtual file system, including its fundamental data structures for file recognition and management. A description of the FUSE framework is given, explaining its architecture, found in both kernel and user space. The extension of file systems in user space is described through some of the implementations it has had in fields such as networks, databases or cloud services, among others. Finally, an initial methodology for implementing a user-space file system using python bindings for the fuse framework and Telegram is described.

Keywords: FUSE library, Linux file system

1 Introduction

The complexity of file systems written in kernel space is constantly growing as new features are implemented. In addition, development is only possible under the C programming language, which handles memory manually and is prone to generate bugs and errors. Therefore, a thorough knowledge of the kernel and its components is necessary, which is quite extensive and time-consuming. As a result, user-space file systems have been seen as a more viable option for different reasons, but mostly because of their ease of development. The FUSE framework is an open source software originally written for Unix and Unix-like operating systems that enables non-privileged users to build file systems in user space.

The goal of this project is to prototype a remote file system in user space using Python, the FUSE framework and Telegram. The implementation would consist of building each of the data structures for file handling and recognition. On top of these data structures, methods will be defined to manipulate them, thus simulating a file system. On the other hand, Telegram self-chats allow uploading an unlimited number of files up to 1.5GB each. The library *telethon* (Telegram's python module), is used to take advantage of this service which would act as the main storage for the file system. The importance of using Telegram self-chats arises for two reasons, to increase our knowledge when working with APIs in Python and to create a cloud-like service by directly storing data in the Telegram service. Thus, this project not only implements the basic concepts of a file system, but seeks a direct application of it by linking it to Telegram.

2 Preliminary Concepts

Although the focus of this report is the implementation of a file system in user space, it is necessary to describe some important concepts for a clear understanding of the subject. In addition, these concepts will be useful later to contrast the differences of a kernel-space file system with respect to user space. The concepts reviewed in this section are the Linux Kernel file system layout and the abstraction layer that allows the execution of system calls known as the Virtual File System.

2.1 Linux File System Architecture

As described by Galloway et al. in [4] the Linux file system is divided into six layers. The application layer, build upon the file access mechanisms of the C Posix library, comprises the user programs being executed as processes. The Portable Operating System Interface often abbreviated as POSIX, is a set of standards specified by the IEEE, aimed to maintain compatibility between operating systems by enabling the source-code portability of applications [3]. The system call interface layer links system resources of the kernel-space to requests performed by applications from the user-space. The Virtual File System layer or VFS layer is an interface for handling and executing system calls by providing the necessary data structures and its implementations. Moreover, it constitutes an environment for the recognition of multiple file systems by executing the necessary processes defined over them. Since application programs could be executed concurrently, the VFS interface should fulfill these possibilities by becoming a concurrent layer. File systems such as FAT, EXT and BTRFs include the necessary instructions to recognize the file system itself making use of the VFS data structures. The individual file systems constitute the fourth layer. Previous to the last layer, resides the buffer cache component, which registers the read and write operations performed by the individual file systems in the storage devices by means of the devices drivers. The deepest layer comprises the device drivers for accessing physical media.

2.2 Virtual File System

Bover and Cesati [1] conceptualize the VFS data structures as the idea of objects in an object-oriented language like C++. An object is a constructor that defines a data structure and all its implementations. Nonetheless, as Linux is build upon C, all of its data types are created using C native structures having specific fields pointing to functions, which could be understood as the implementations of that specific object. A description of the VFS data structures is given hereafter.

– **Superblock Object.** Contains all the information of a mounted file system such as type, id of the physical device, total size and a pointer to the root directory entry.

- **Inode Object.** Records the metadata of a file such as type (regular file, directory or device), permissions, callbacks to its file system and its mapping in memory. An inode object is given a unique inode number capable of recognizing a file in the file system.
- **File Object.** Contains metadata about the interaction of an open file and a process. This information is available only in kernel-mode as long as the process requires the file to be opened.
- **Dentry Object.** Handles the metadata of a directory entry (filename) and its linking to a corresponding file. Some of its fields are name, reference to the parent directory, a list of sub-directories and contiguous directories, mount information, reference to the corresponding superblock, reference to the corresponding inode and a counter to know the number of process it is hosting.

Consider the following example illustrated in Figure 1. In this case, three processes read the same file, the first process uses one hard link whereas process three and two share a second hard link to the file. A hard link is a directory entry that associates a filename to its inode. Because, there are only two hard links, then only two dentry objects are needed to associate the file name to its inode. As both dentry objects refer to the same inode, this inode recognizes the superblock object which allows to access the required file in disk.

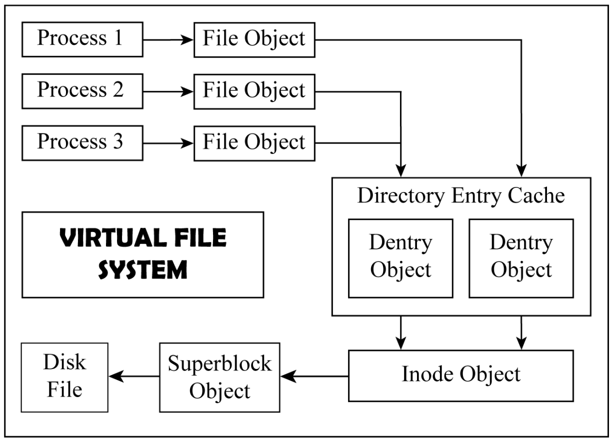


Fig. 1. Instance of the Virtual File System. Adapted from Bover and Cesati [1].

3 File System in User Space

Zadok and Badulescu in [12], describe the Linux file system as a kernel interface that enables data storage and retrieval by interacting directly with lower-level media, such as disks and networks. Hence, a deep understanding of the kernel and its components is required, with the C programming language being the only accepted language for kernel code development. The file system interacts not only with the virtual file system in response to system calls made from user space, but also with the virtual memory handler for page allocation and memory management in the kernel and with the virtual device layer for handling storage devices and data storage. Vangoor et al. in [10], explain that contemporary file systems such as BTRFS contains more than 85000 lines of code, but could reach up to millions in comparison to other file systems such as IBM GPFS. As it is observed, kernel file systems have a high complexity to develop as a result of the common interactions they must provide, but also because of additional functionalities offered such as B-tree search, access control lists (ACLs), encryption and others.

By contrast, programming in the user space minimizes the complexity observed in kernel space, as a wider range of tools is available to developers. As a result, multiple libraries have been developed for writing file systems in user space, the FUSE library being the most notorious one. Narayan et al. [8], explain that FUSE allows non-privileged users to create file systems by transferring system calls into the user space. However, this is achieved at the expense of increased context switching and memory copies. Even so, Vangoor et al. in [10] and [11], explain that FUSE has managed to attract attention due to the following reasons:

- **Development Ease.** Advantage of tracing, debugging and profiling tools. A crashing program is able to generate an error report. Development is no longer limited to C.
- **Portability.** It is easier to port the file system to a different platform, since it is not developed for a specific kernel.
- **Libraries.** The performance of a file system in user space could be improved by using more efficient algorithms. For example, AI algorithms could be implemented to improve predictive prefetching.
- **Increased Functionality.** Stackable file systems increase the functionality of kernel file systems by implementing techniques such as deduplication and compression.

3.1 FUSE Framework

As explained by Rajgarhia and Gehani [9], the FUSE framework provides both a low-level API and a high-level API. Developing a file system under the low-level API requires the user to handle inodes and pathname translations, including caching. In addition, the fields of the data structure for executing system calls is loaded by the file system. On the contrary, the high-level API just requires the developer to handle pathname translations and loading the fields of the system calls data structures. In comparison, the low-level API behaves like the Virtual File System, while the high-level API behaves like the interface for executing system calls which makes it easier to implement.

Despite the selected API, the FUSE framework has a high-level architecture that consists of a FUSE kernel module, *fuse.ko* and a FUSE daemon at the user space level. In addition, and although it is not a component of the architecture, the library that contains all of the implementations or methods of FUSE to execute operations is libfuse. Vangoor et al. [10], explain that when the *fuse.ko* module is initialized within the kernel, three types of file systems are available recognized in the VFS: *fuse*, *fuseblk* and *fusectl*. The *fuse* file system do not requeires a specific device partition as it can be stacked into memory or network file systems. The second type, *fuseblk* file system, behaves as a traditional local file system as is mounted in a specific block of a storage device. At last, the tools to handle the functionality of a FUSE file system reside within the *fusectl* file system; for instance, registering the number of pending requests.

A description of the FUSE file system and its operations follows based on the concepts illustrated by Vangoor et al. in [10] and Rajgarhia and Gehani in [9].. The execution of a mapped application on a FUSE file system is registered by the Virtual File System which, in turn, handles the request to the FUSE kernel driver. The FUSE request is inserted into a FUSE queue and remains in a waiting state until the FUSE daemon, which resides in user space, can handle the request. The FUSE daemon will perform the request by reading from the partition assigned to the *fuse* file system, normally being */dev/fuse*. Making the request requires the FUSE daemon to return to kernel space, but the condition depends on the type of FUSE file system. In the case of a stackable file system, the daemon communicates with an underlying kernel file system such as EXT3 or EXT4. For a block device file system, the FUSE daemon executes write and read operations on that block. Finally, for an in-memory or network file system, the daemon might request system services such as socket creation or time-of-day synchronization. Once the request completes, the daemon sends a response to */dev/fuse*. Control is then returned to the FUSE kernel driver, which will update the status of the request to completed and reset the user process to continue its execution.

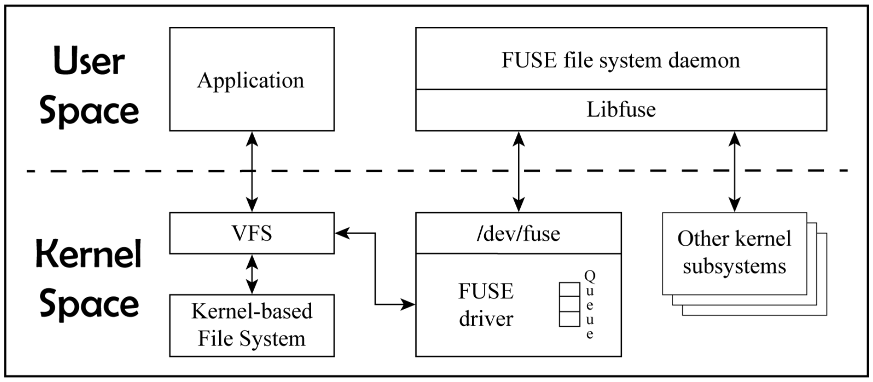


Fig. 2. Fuse Architecture. Adapted from Vangoor et al. in [10]

4 Related Work

The following tables give a clear idea of how extensive the FUSE framework has become over the years by showing some of its application in different fields.

Name	*Author*	*Usage*	*Description*
No Prototype	Alexander Holupirek and Marc Scholl	Implements databases with XML to build file systems in user space.	They added XML as the third type of file and exposed the content of the files, with the structure included, to the system and user that was initially hidden.[6]
fusexmp	Miklos Szeredi and Tejun Heo	Mirrors the root directory of the system on the mounted directory.	It is widely used in example tests since it uses a local directory tree as the backing store for the FUSE mount.
BGZIP	Ronnie Sahlberg	An overlay file system that decompresses indexed GZIP files on the fly.	Allows to access the content of large GZIP files without first having to uncompressed them.
ChunkFS	Florian Zumbiehl	Supports mounting an arbitrary file or block device as a file directory tree.	It was written to make incremental, space-efficient backups of encrypted file system images using rsync.
RAIF	Nikolai Joukov, et al.	Allows the user to define rules to determine the data placement policy. It also allows users to distribute data in different file systems and define redundancy in them.	Because RAIF is a fan-stackable file system, it works by overlaying new functionality on top of existing file systems. It uses another file system instead of performing operations on a backup store such as a disk. [7]
1fichierfs	Alain Bénédetti	It is a specialization of astreamfs for 1fichier cloud storage.	Using the API provided by 1fichier, the user no longer needs the script, it is only necessary to provide the API key and the mount point.
UmbrellaFS	Garrison, John A. and Reddy, A. L. Narasimha	Allows users to define distribution policies between different devices, each with its own redundancy and performance limitation.	Inserts another layer in the kernel, above the host file system and below the Virtual File System. Thus, it is possible to place files from several file systems in the same namespace. [5]

5 Methodology

The goal of this project is to implement a remote user space file system that uses Telegram self-chats as the primary storage. The idea of the project is based on the guide created by Davoli et al. [2] in order for students in an Operating Systems class to implement a file system in user space, which is normally taught theoretically. Since file system operations can execute concurrently and asynchronous, the guide begins by giving a brief introduction to asynchronous programming and how it is implemented in Python. It goes on to explain the basic operations that are executed on a file and how to interact with Telegram's API by creating a client to access the service. The rest of the guide is devoted to implementing the file system in user space by making use of Python, FUSE and Telegram. Finally, the authors explain some implementation issues and different enhancements to improve performance.

Two environments are being used for implementing and testing the developed file system, their description is presented in Table 1. The file system will be built using Python 3.10, the *pyfuse3* library which contains the *fuse3* framework bindings for python, the *asyncio* library which allows asynchronous programming and the *telethon* library which is a Telegram client module for Python 3 that allows connecting to the public API. The file system will be built upon the following four aspects:

1. Data Structures: Inode, Superblock, DirectoryData and FileData.
2. Wrapper: connection, helpers, serialization and public API.
3. Script for building an empty filesystem with a specific number of inodes and only one root directory.
4. File system methods: operations, initialization and mounting.

#	*Linux Distribution*	*Virtualization Environment*	*RAM*	*CPU*	*Libraries*
1	Arch Linux	None	16GB	i7-9750H	fuse3, pyfuse3, telethon, asyncio
2	Ubuntu	VMware Workstation Player	3GB	i5-10300H	fuse3, pyfuse3, telethon, asyncio

Table 1. Description of each Implementation Environment

5.1 Data Structures

The data structures were implemented as classes having the necessary methods for its manipulation within the file system. A description of each data structure and its methods is presented in Table 2.

Data Structures	*Description*	*Methods*
Superblock	The superblock contains all the information of the mounted file system. It stores a list of all inodes and has the methods to create, delete and fetch for an inode.	– *__init__(self, number)* – *get_new_inode(self, number)* – *free_inode(self, number)* – *get_inode_by_number(self, number)*

Data Structures	*Description*	*Methods*
Inode	It recognizes either a regular file or a directory. A reference to the file stored as a message in the self-chat is kept by fetching the id of the message.	– *__init__(self, number)* – *is_directory(self)* – *is_regular_file(self)*
DirectoryData	Stores the metadata of directory entries keeping a reference of both its parent inode and its current directory inode.	– *__init__(self, self_inode_n, parent_inode_n)* – *__len__()*
FileData	Stores the contents of a file in bytes and is initialized as an empty string of bytes.	– *__init__(self, initial_data=b'')* – *__len__(self)*

Table 2. Description of the data structures

5.2 Wrapper

Recalling that the Telegram self-chat service will be used as the file system storage, a number of functionalities were designed in order to interact with the service. As a result, the class *TgFuseWrapper* was defined, which is initialized with a phone number associated with a Telegram account. Its method *__init__()* will establish the client that enables the connection to the Telegram service. The description of the three main functionalities designed for this class and their methods are presented in Table 3, optional parameters are not presented.

TgFuseWrapper Class		
Functionality	*Description*	*Methods*
Superblock methods	The superblock and its features are defined within a file that would be send and pinned in the self-chat acting as the storage device. The superblock file is serialized in order to be uploaded in the self-chat. If its contents are required, then it is first downloaded and then deserialized.	– *write_superblock(self, superblock)* – *read_superblock(self)*
Serialization	To create or update changes to a file or directory, the object must first be serialized. Serializing the file or directory object allows it to be in a format that can be sent as a message. Similarly, to read the contents of a file or directory, it is first searched within the self-chat and, if found, deserialized and returned. The serialization or deserialization of the file or directory object was obtained using the *pickle* library.	– *_pickle_and_save(self, obj)* – *_unpickle(file)*

TgFuseWrapper Class		
Functionality	*Description*	*Methods*
Files and directories manipulation methods	This functionality ensures operations of creation, deletion, reading and modification of files saved in the mounted file system within the self-chat. Moreover, it also provides the routines in order to upload and download a file or directory object.	– *write_data(self, data)* – *delete_data(self, message_id):* – *read_data(self, message_id)* – *_upload_data(self, data)* – *_download_data(self, message_id)*

Table 3. TgFuseWrapper Class

5.3 File System building Script

The next aspect implemented was the script presented in Listing 1.1, which allows to create an empty file system capable of handling 1000 inodes with the first inode id assigned to the root directory. The root directory is created using the *DirectoryData* class presented in Table 2. The class *DirectoryData* is initialized with two parameters which are a pointer to the directory inode itself and a pointer to the parent directory inode. The root directory stores all files and sub directories of the file system, and as such both pointers are a reference to the same inode as observed in line 21 of Listing 1.1. An important feature of this script, is that it allows to restart the file system if it has been corrupted or if necessary.

```
 1 import asyncio
 2 from stat import S_IFDIR
 3 from time import time_ns
 4 from data_structures import *
 5 from wrapper import *
 6 
 7 
 8 async def make():
 9     s = Superblock(1000)
10     inode = s.get_inode_by_number(1)
11 
12     entry = inode.attributes
13     entry.st_mode = (S_IFDIR | 0o755)
14     stamp = time_ns()
15     entry.st_atime_ns = stamp
16     entry.st_ctime_ns = stamp
17     entry.st_mtime_ns = stamp
18     entry.st_gid = os.getgid()
19     entry.st_uid = os.getuid()
20 
21     dd = DirectoryData(1, 1)
22     entry.st_size = len(dd)
23     message_id = await w.write_data(dd,'ROOT directory data')
24     inode.data_pointer = message_id
25     await w.write_superblock(s)
26 
27 w = TgFuseWrapper(sys.argv[1])
28 asyncio.get_event_loop().run_until_complete(make())
29 
```

Listing 1.1. Empty File System Script

5.4 File System Operations

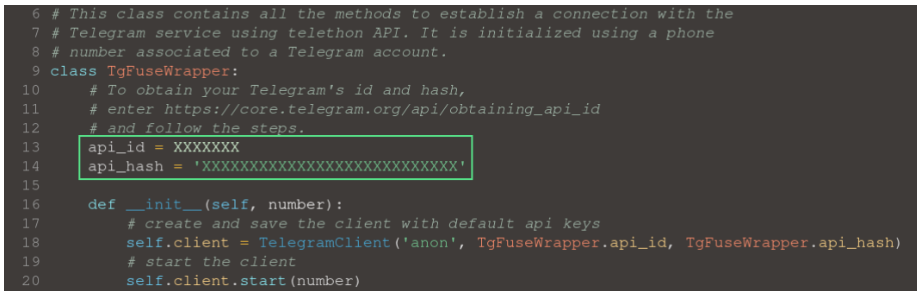
The last aspect implemented is the operations offered by a common file system. In this sense, the file system must provide routines for creating, deleting, writing, reading, opening and closing files and directories, as well as establishing the times of their creation or modification. Furthermore, a lookup method should be included to explore the contents of a directory and determine whether a file exists or not. Finally, a deletion of a file or directory implies that the inode id granted must be released and dereferenced from the superblock. The operations implemented for the file system belong to the *TgFuseFs* class and are presented in Table 4 making a distinction for those destined to files, directories and its removal.

Type	*Methods*
File	– *_create(self, parent_inode_n, name, mode, ctx)* – *open(self, file_inode_n, flags, ctx)* – *read(self, fh, off, size)* – *write(self, fh, off, buf)* – *release(self, fh)*
Directory	– *_lookup(self, parent_inode_n, name, ctx=None)* – *opendir(self, inode_n, ctx)* – *readdir(self, fh, start_id, token)* – *releasedir(self, fh)* – *_update_directory(self, dir_ino_n, entries)* – *close(self)*
Removal	– *unlink(self, parent_inode_n, name, ctx)* – *_remove(self, parent_inode_n, name, ctx, is_dir=False)* – *forget(self, inode_list)*

Table 4. TgFuseFs Class

6 Results and Discussion

Hereinafter, an explanation of every step to deploy an use the file system is presented. Each of the following instructions considers that the user has a Telegram account and has obtained an *api id* and a *api hash* to be able to establish the client that will enable the use of Telegram's API. This parameters must be added into the two static constants of the *TgFuseWrapper* class as observed in Figure 3.

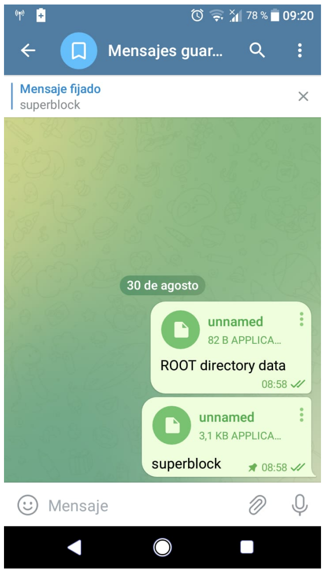


```python
# This class contains all the methods to establish a connection with the
# Telegram service using telethon API. It is initialized using a phone
# number associated to a Telegram account.
class TgFuseWrapper:
    # To obtain your Telegram's id and hash,
    # enter https://core.telegram.org/api/obtaining_api_id
    # and follow the steps.
    api_id = XXXXXXX
    api_hash = 'XXXXXXXXXXXXXXXXXXXXXXXXXXXX'

    def __init__(self, number):
        # create and save the client with default api keys
        self.client = TelegramClient('anon', TgFuseWrapper.api_id, TgFuseWrapper.api_hash)
        # start the client
        self.client.start(number)
```

Fig. 3. Set the api id and api hash within the class *TgFuseWrapper*.

Consequently to create the empty file system with the root directory and superblock the *mktgfuse.py* is executed using the command *–python3 mktgfs.py +593xxxxxxxxx–* with the third argument being the phone number. The previous command considers that the phone number belongs to Ecuador, otherwise the international country code must be changed such as *+56xxxxxxxxxx* for Chile or *+52xxxxxxxxxx* for Mexico. After the command is executed, two files will appear in the user's Telegram self-chat, one named *ROOT directory data* and the other being *superblock* as observed in Figure 4.

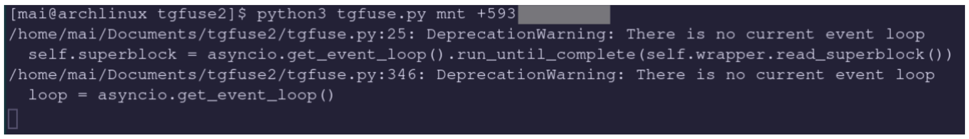


unnamed
82 B APPLICA...
ROOT directory data
08:58

unnamed
3,1 KB APPLICA...
superblock
08:58

Fig. 4. Empty file system initialization

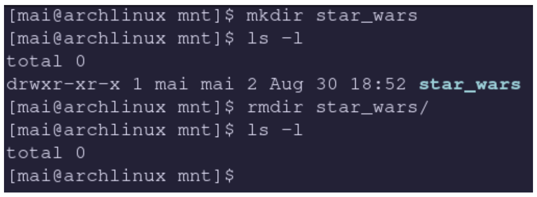
Subsequently a directory must be created to mount the file system within the computer, this directory would act as the root directory of the file system. This allows to create files and directories that would be saved in Telegram's self-chat which acts as the storage device of the file system. To mount the file system the commands –*mkdir mnt*– followed by –*python3 tgfuse.py mnt +593xxxxxxxxx*– are executed, with the output message of the second command being shown in Figure 5. After executing the second command the terminal should remain active to maintain a connection with Telegram's API.



```
[mai@archlinux tgfuse2]$ python3 tgfuse.py mnt +593[redacted]
/home/mai/Documents/tgfuse2/tgfuse.py:25: DeprecationWarning: There is no current event loop
  self.superblock = asyncio.get_event_loop().run_until_complete(self.wrapper.read_superblock())
/home/mai/Documents/tgfuse2/tgfuse.py:346: DeprecationWarning: There is no current event loop
  loop = asyncio.get_event_loop()
```

Fig. 5. Output message after executing the tgfuse.py file

Once the file system is mounted, Linux-like operations can be performed for creating (*mkdir, touch*) and deleting (*rmdir, rm*) directories and files. In addition, methods such as *ls [dir_name]* to list the contents of a directory and *cd [dir_name]* to change the directory are also available. Creating a file or directory within the *mnt* directory will show a new uploaded file in the self-chat. Although the uploaded file has no caption, the superblock file contains the *inode_id* to recognize each of the created files or directories. For example, creating a directory named *star_wars*, Figure 6, would appear as an unnamed file in the self-chat as shown in Figure 7. Conversely, if the directory is deleted, Figure 6, the file representing the directory in the self-chat must also be removed; such behavior is depicted in Figure 8. Although not presented, the same results are observed for file creation and deletion using the commands *touch* and *rm* respectively.



```
[mai@archlinux mnt]$ mkdir star_wars
[mai@archlinux mnt]$ ls -l
total 0
drwxr-xr-x 1 mai mai 2 Aug 30 18:52 star_wars
[mai@archlinux mnt]$ rmdir star_wars/
[mai@archlinux mnt]$ ls -l
total 0
[mai@archlinux mnt]$
```

Fig. 6. Creation and removal of the *star_wars* directory inside *mnt*

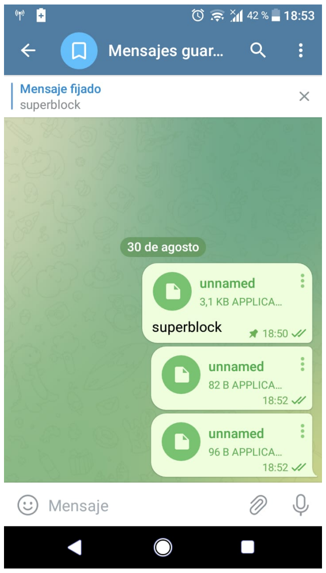


Fig. 7. Creation of the *star_wars* directory within the self-chat

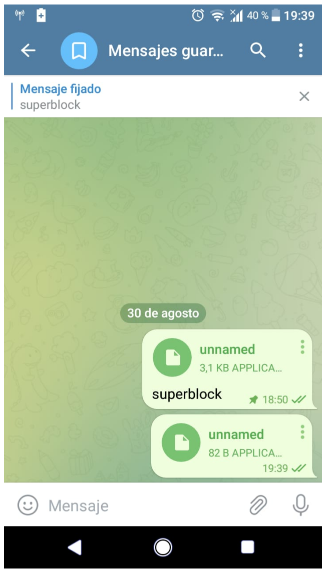


Fig. 8. Removing the *star_wars* directory from the self-chat.

7 Conclusions

Although there is a noticeable reduction in performance due to increased context switching and memory copies when using a file system in user space, the advantages it offers over development in kernel space make it easier to manage. Its scope has not only been targeted to a specific system or purpose, but has been extended to fields such as networks, databases, cloud services, compressed files, storage policies, among others. As a result, different programming languages have developed bindings for the FUSE framework, allowing a known syntax to be used when creating file systems in user space. In this context, and taking into account cloud storage services, the goal of this project was to use Python to build a user space file system, while interfacing with Telegram, where its self-chat service is used as the main storage. Leveraging Python's object-oriented paradigm, data structures for file recognition and management were created and represented as classes. The prototype shown in this work is just a demonstration of how a file-system can be written and how it works by using the Telegram framework. With the time and evolution of better technologies, developers will be able to create more efficient and applicable file-systems. This implementation, however, is far from being perfect; there are issues related to the maximum data storage allowed by Telegram, and performance due to the simple locks used to avoid race conditions. Nevertheless, this implementation is suitable for academic and teaching purposes since the prototype works properly.

References

1. Bover, D.P., Cesati, M.: The Virtual Filesystem, p. 456–458. O'Reilly, third edn. (2006)
2. Davoli, R., Sbaraglia, D.M., Lodi, D.M., Maffei, R.: Tgfusefs: How high school students can write a filesystem prototype
3. Gallmeister, B.: POSIX. 4 Programmers Guide: Programming for the real world. " O'Reilly Media, Inc." (1995)
4. Galloway, A., Lüttgen, G., Mühlberg, J.T., Siminiceanu, R.I.: Model-checking the linux virtual file system. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 74–88. Springer (2009)
5. Garrison, J.A., Reddy, A.L.N.: Umbrella file system: Storage management across heterogeneous devices. ACM Trans. Storage **5**(1) (mar 2009). https://doi.org/10.1145/1502777.1502780, https://doi.org/10.1145/1502777.1502780
6. Holupirek, A., Scholl, M.: An xml database as filesystem in userspace. Publ. in: Proceedings of the 20. GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), Apolda, Thüringen, Germany, May 13-16, 2008, pp. 31-35 (01 2008)
7. Joukov, N., Krishnakumar, A.M., Patti, C., Rai, A., Satnur, S., Traeger, A., Zadok, E.: RAIF: redundant array of independent filesystems. In: 24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007), 24-27 September 2007, San Diego, California, USA. pp. 199–214. IEEE Computer Society (2007). https://doi.org/10.1109/MSST.2007.30, http://doi.ieeecomputersociety.org/10.1109/MSST.2007.30
8. Narayan, S., Mehta, R.K., Chandy, J.A.: User space storage system stack modules with file level control. In: Proceedings of the 12th Annual Linux Symposium in Ottawa. pp. 189–196. Citeseer (2010)
9. Rajgarhia, A., Gehani, A.: Performance and extension of user space file systems. In: Proceedings of the 2010 ACM Symposium on Applied Computing. pp. 206–213 (2010)
10. Vangoor, B.K.R., Agarwal, P., Mathew, M., Ramachandran, A., Sivaraman, S., Tarasov, V., Zadok, E.: Performance and resource utilization of fuse user-space file systems. ACM Transactions on Storage (TOS) **15**(2), 1–49 (2019)
11. Vangoor, B.K.R., Tarasov, V., Zadok, E.: To {FUSE} or not to {FUSE}: Performance of {User-Space} file systems. In: 15th USENIX Conference on File and Storage Technologies (FAST 17). pp. 59–72 (2017)
12. Zadok, E., Badulescu, I.: A stackable file system interface for linux. In: LinuxExpo Conference Proceedings. vol. 94, p. 10 (1999)